3
O Framework *FLoCS*

Neste capítulo propomos um framework que serve como um arcabouço genérico para serviços de comunicação baseada em localização.

Inicialmente, apresentamos alguns conceitos básicos relacionados com o trabalho. Na Seção 4.2, apresentamos uma visão geral do framework e na seção subseqüente descrevemos os aspectos técnicos de sua arquitetura.

3.1.
Fundamentação Conceitual

A criação de um arcabouço genérico para a construção de novos aplicativos pressupõe o conhecimento de duas técnicas de reuso de software: *frameworks* e *padrões de projeto*.

3.1.1.
Frameworks

Um *framework* pode ser definido como um projeto genérico em um domínio que funciona como um molde para a construção de aplicações ou subsistemas específicos (Souza et al., 1998).

Todas as aplicações construídas a partir de um mesmo framework apresentam a mesma estrutura, diferenciando-se no seu comportamento, que é dependente da aplicação. Nos frameworks, a estrutura da aplicação e o fluxo de execução já estão pré-definidos. Para que suas funcionalidades sejam estendidas, é necessário apenas reescrever alguns métodos de determinadas classes ou implementar algumas interfaces. Dessa forma, ele é adaptado segundo as necessidades de uma aplicação específica.

A utilização de frameworks oferece uma série de vantagens à atividade de desenvolvimento de software. A principal é a possibilidade de reutilização de código e projeto, o que torna o desenvolvimento de novas aplicações mais rápido e barato, uma vez que emprega componentes que já foram utilizados e testados por outros desenvolvedores.

De uma maneira geral, para a construção de um framework, segue-se a metodologia desenvolvida por (Pree,1994), que se baseia na identificação dos *pontos fixos (frozen spots)* e dos *pontos adaptáveis* ou *flexíveis (hot spots)* do domínio de uma aplicação. Os pontos fixos correspondem às partes comuns das aplicações, enquanto os pontos adaptáveis são as partes que podem ser estendidas para cada aplicação específica. Esses pontos adaptáveis dão ao framework a capacidade de ser flexível, moldando-o a diferentes aplicações.

Para a implementação de frameworks, uma boa prática é o uso de *padrões de projeto*, que serão apresentados a seguir.

3.1.2. Padrões de Projeto

Padrões de projeto (Gamma et al., 1994) podem ser definidos como soluções reutilizáveis para problemas que se repetem em contextos similares durante o desenvolvimento de software (Grand, 1998).

Eles têm sido propostos como um meio de representar, registrar e reutilizar micro-arquiteturas de projeto comuns e recorrentes, bem como a experiência acumulada por projetistas ao desenvolver essas estruturas. Com o conhecimento desses padrões, os programadores podem reconhecer as situações em que cada padrão pode ser aplicado e imediatamente usar a solução, sem ter que analisar o problema e tentar diferentes estratégias para resolvê-lo.

Alguns padrões de projeto foram utilizados repetidas vezes no projeto do framework proposto. De forma a tornar mais clara a descrição do framework, segue um breve resumo dos principais padrões utilizados.

- *Singleton*

Este padrão é utilizado quando se necessita que haja no sistema exatamente uma única instância de uma determinada classe acessível a todos os seus clientes.

Singleton assegura que apenas uma única instância de uma classe é criada. Assim, todos os objetos que precisam usar uma instância dessa classe, usam a mesma instância.

Para sua implementação, deve-se criar uma variável estática que guarde a instância única. Para que esta seja acessada, a classe deve prover um método estático que retorne uma referência à variável estática. Além disso, deve-se implementar todos os construtores da classe como privados para que não seja possível criar nenhuma instância adicional. Outras classes que queiram acessar a instância única têm de chamar o método estático e não criar uma instância da classe através de seu construtor.

- *Factory Method*

Em alguns sistemas, o conjunto de classes que uma dada classe necessita instanciar pode não estar definido a priori, inclusive podendo variar à medida que novas classes forem incluídas/implementadas.

Factory Method resolve esse problema permitindo que uma classe possa ser independente das classes que instancia. Para tal, delega-se a escolha de qual classe será instanciada a outro objeto, chamado de fábrica, e referenciando-se o objeto criado através de uma interface comum. A classe principal somente passa para a fábrica uma string com o tipo de objeto a ser criado, e a fábrica lhe retorna tal objeto.

- *Adapter*

Esse padrão é utilizado quando se tem uma classe que chama um método através de uma interface e deseja-se que tal classe chame um método de um objeto que não implementa essa interface. Por exemplo, essa situação ocorre com freqüência quando um programa precisa ter acesso a sistemas legados.

Adapter utiliza como solução uma classe adaptadora que implementa a interface chamada pela classe cliente. Seus métodos chamam os métodos relacionados na classe adaptada, fazendo as adaptações necessárias.

3.2. Visão geral do framework

Com base na análise de algumas aplicações para comunicação baseada em localização e a experiência adquirida no desenvolvimento de uma delas, o *Nita*, propomos nesta dissertação um framework para instanciar aplicações desse tipo, que chamamos de *FLoCS (**F**ramework for **Lo**cation-based **C**ommunication **S**ervices)*.

Com o estudo do domínio das ACBL, foram separados os aspectos comuns do domínio dos específicos de cada aplicativo. Os aspectos comuns deram origem aos pontos fixos *(frozen spots)* do framework e os específicos, aos pontos adaptáveis ou flexíveis *(hot spots)*. O conjunto de pontos fixos provê uma infra-estrutura básica a todas as aplicações instanciadas a partir do framework, enquanto que os pontos adaptáveis são compostos de classes abstratas ou interfaces, com métodos mais genéricos possíveis. Na instanciação do framework, tais classes abstratas ou interfaces darão origem a classes concretas que especializarão as funcionalidades existentes e possivelmente adicionarão novas, implementando dessa forma os detalhes específicos de cada aplicação. Todos os pontos adaptáveis do *FLoCS* têm uma ou mais implementações padrão que podem ser reutilizadas pelo desenvolvedor da aplicação.

As características principais do framework são apresentadas a seguir e serão descritas com mais detalhes na próxima seção.

- **Arquitetura**

Os serviços de comunicação baseada em localização são geralmente compostos por clientes, executando em dispositivos móveis, e um servidor, na rede fixa. O foco do framework proposto é a instanciação do servidor da aplicação, onde concentra a maior parte dos serviços e estabelece uma arquitetura que os interconecta. Para o cliente, o framework oferece apenas serviços de comunicação e persistência de dados e não impõe nenhuma arquitetura, cabendo ao desenvolvedor criar a interface gráfica e a lógica da aplicação que utiliza tais serviços.

- **Comunicação**

Como os protocolos para a comunicação entre os clientes e o servidor podem variar muito dependendo da aplicação, este é um ponto adaptável do framework. Este *hot spot* faz parte do serviço de comunicação que, entre outras funcionalidades, permite que uma aplicação escolha, em tempo de execução, qual protocolo de comunicação deseja utilizar. O framework oferece duas implementações para esse ponto adaptável: uma que provê comunicação por meio de sockets TCP e outra que serve como interface para um *serviço de eventos (publish/subscribe)*.

FLoCS possibilita que os usuários das aplicações instanciadas comuniquem-se por meio de mensagens, tanto de forma síncrona quanto assíncrona. A comunicação assíncrona é provida através da associação de mensagens a localidades simbólicas. Os destinatários só receberão as mensagens quando estiverem na localidade onde estas foram postadas. Para prover comunicação síncrona, o framework garante o envio da mensagem diretamente para o usuário, sem esperar por nenhuma pré-condição.

- **Posicionamento**

A tecnologia de localização utilizada para obter a posição de uma dispositivo móvel é outro ponto adaptável do *FLoCS*. O serviço de posicionamento do framework permite que as aplicações possam utilizar tanto tecnologias de localização centradas na rede quanto as centradas no dispositivo. Um outro ponto adaptável do serviço de posicionamento é o gerenciador de regiões simbólicas, que mapeia os dados sobre a posição geográfica fornecidos pela tecnologia de localização para regiões simbólicas.

- **Persistência de Dados**

As aplicações necessitam de algum mecanismo para persistirem seus dados, mensagens, preferências dos usuários, regiões simbólicas, etc. Como tais mecanismos podem variar desde o uso de simples sistemas de arquivo a sofisticados bancos de dados, este também é um ponto adaptável do framework. *FLoCS* oferece como implementação padrão um serviço de persistência baseado em arquivos.

- **Localidades**

Grande parte das aplicações considera as localidades simbólicas apenas como entidades passivas, responsáveis somente por guardar as mensagens postadas nelas. No *FLoCS,* entretanto, tais entidades são geradoras de eventos e têm diversos atributos que podem ser estendidos através de um objeto de configuração, que é um ponto adaptável do framework. Além disso, a cada localidade está associado um agente de software (Maes, 1995) responsável pela exclusão das mensagens cujo tempo de validade expirou. O controle do ciclo de vida e gerenciamento das localidades fica a cargo de um gerente de localidades.

- **Eventos**

Uma importante característica do framework é permitir que quaisquer objetos possam registrar interesse em eventos, tais como mudança de localização de um usuário, postagem de uma mensagem em uma localidade e outros. Quando ocorre um evento, o serviço emissor (por ex., a mudança de localização de um usuário é um evento disparado pelo serviço de posicionamento) fica responsável por notificar os objetos interessados. A interação através de eventos foi usada de forma a prover um desacoplamento entre o emissor e o receptor, permitindo que vários objetos efetuem ou cancelem seus registros a qualquer tempo, provendo maior flexibilidade às aplicações.

3.3. Arquitetura

Esta seção apresenta a arquitetura do framework, demonstrando como suas características apresentadas na seção anterior foram implementadas. As classes foram agrupadas em pacotes de acordo com suas funcionalidades. Essa forma de particionar o framework permite que, na sua instanciação, o desenvolvedor se concentre em apenas um aspecto de cada vez. A Figura 1 ilustra o particionamento do framework.

A seguir, serão descritos os componentes de cada pacote e como eles interagem entre si. Depois, alguns casos de uso serão usados para descrever a dinâmica do framework.

3.3.1. Comunicação

O pacote *flocs.communication* reúne todas as classes responsáveis pelo envio e recebimento de dados entre o cliente e o servidor. A forma de comunicação é um ponto adaptável do framework. Tal *hot spot* é representado pela interface *CommunicationIF,* cujos métodos são flexíveis suficientes, permitindo integrar ao framework várias formas de comunicação.

A classe principal do pacote é *CommunicationService (CS)* que serve como fachada para as diversas classes de comunicação. Seu método *send* permite enviar qualquer tipo de dado, desde que este implemente a interface *java.io.Serializable* de Java. O método *send* também recebe como parâmetro o tipo de comunicação que será utilizado, possibilitando que a comunicação entre o cliente e o servidor seja feita de diversas formas. Por exemplo, o comando de autenticação do usuário pode ser feito por socket TCP e a postagem de mensagens através de um serviço de eventos do tipo *publish/subscribe.*

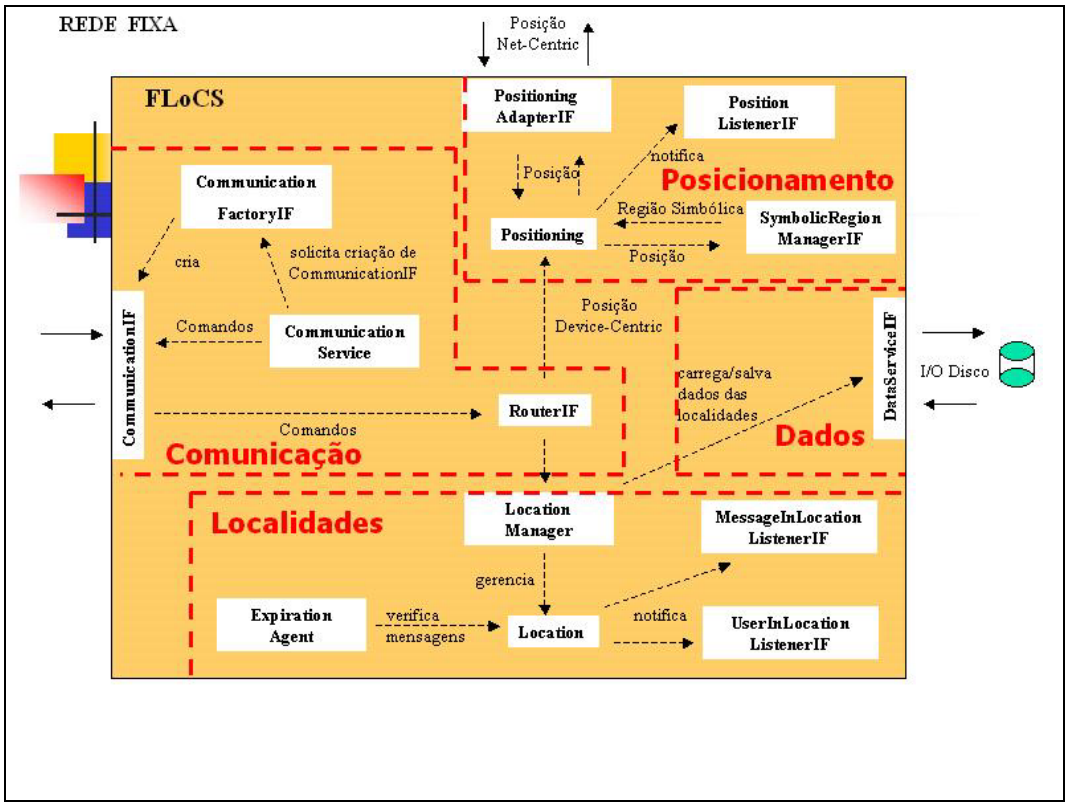

Figura 1 – Particionamento do framework.

Para que seja possível instanciar outras classes de comunicação sem que *CS* seja dependente destas, foi usado o padrão de projeto *Factory Method.* Assim, quando um método de *CS* é chamado, este requisita à classe que corresponde à fábrica do padrão o tipo de comunicação desejado e chama neste último o método correspondente.

A fábrica do padrão de projeto é representada pela interface *CommunicationFactoryIF,* que o desenvolvedor deve implementar de modo que seu método *createCommunication* retorne a classe que implementa o tipo de comunicação recebido como parâmetro.

Para que a aplicação obtenha os comandos que chegam pela rede, alguma de suas classes deve implementar a interface *RouterIF*. Esta possui apenas um único método que é chamado por *CommunicationIF* toda vez que este receber um comando. *RouterIF* faz o papel de roteador, pois esta classe recebe o comando, verifica seu tipo e o redireciona para a classe do sistema que vai tratá-lo. Por exemplo, quando chega um comando de *login*, este será redirecionado para uma classe que faça a autenticação de usuários. Geralmente, as classes que implementam *Router*IF têm a função de classe controladora, definindo o fluxo de comandos no sistema.

O framework oferece duas classes de comunicação, que implementam a interface *CommunicationIF*: sockets TCP, por meio da classe *TcpSocket,* e uma fachada para o serviço de eventos *ECI*, da arquitetura *Moca* (Sacramento et al., 2004), através das classes *MocaPubSubClient* e *MocaPubSubServer*. O motivo para a existência dessas duas últimas classes é devido a uma idiossincrasia do *ECI* que determina programações diferentes para o módulo cliente e o servidor. Entretanto, como essas diferenças não são importantes do ponto de vista do framework, doravante nos referiremos a essas duas classes como se fossem uma única, que chamamos de *MocaPubSub.*

Os desenvolvedores que necessitarem de outros tipos de comunicação devem implementar a interface *CommunicationIF*. Nessa interface, há dois métodos para iniciar/parar a *thread* de comunicação, que fica constantemente monitorando a requisição por novas conexões, outros dois métodos para registrar/desregistrar um destino (por exemplo, o *host* do servidor ou do cliente móvel), e finalmente um método para enviar dados.

O método *init*, que inicia o servidor, recebe dois parâmetros: a porta na qual o servidor vai se conectar e uma referência para *RouterIF*, para que seu método *onCommand* seja chamado quando chegar algum comando do cliente.

O método *registerDestination* recebe dois parâmetros: uma string que representa o destino e um objeto para encapsular as informações necessárias para abrir uma conexão com este destino. Tal objeto, por ser do tipo *Object*, pode ser qualquer tipo de objeto definido pelo desenvolvedor. Isso porque cada tipo de comunicação exige configurações diferentes. Por exemplo, na classe *TcpSocket,* o objeto passado é do tipo *NetInfo*, que é uma classe que encapsula o endereço IP e a porta do cliente, informações necessárias para estabelecer uma conexão TCP. Registrar um destino é indexar as informações necessárias para abrir uma conexão com tal destino através de um identificador. Assim, essas informações são passadas apenas uma vez e não sempre que se for enviar um dado.

Por fim, o método *send* recebe como parâmetro o comando e a string que identifica o destino.

O framework oferece também a classe *CommunicationFactory*, uma implementação para *CommunicationFactoryIF*, que cria objetos *MocaPubSub* e *TcpSocket.*

A Figura 2 ilustra o diagrama de classes deste pacote.

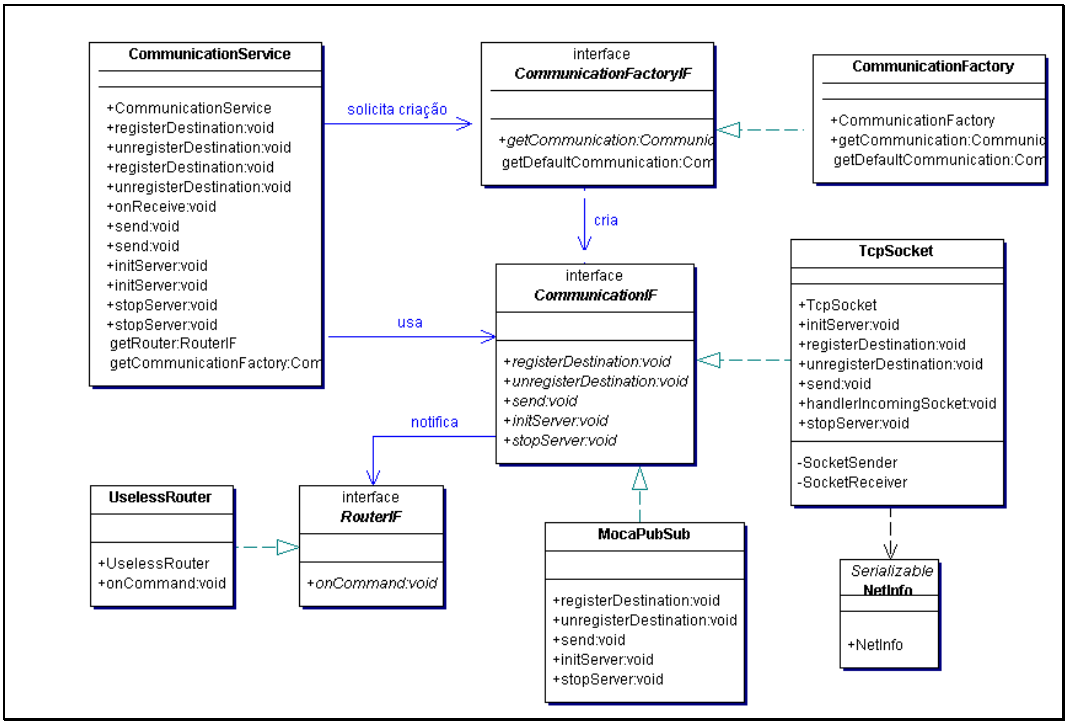

Figura 2 – Diagrama de classes do pacote *flocs.communication.*

3.3.2. Posicionamento

O pacote *flocs.positioning* contém as classes responsáveis em prover a localização de um dispositivo móvel, que deve ocorrer através de alguma tecnologia de localização. Sua classe principal é *Positioning,* que serve de fachada entre a aplicação e as outras classes deste pacote, que fornecem informação de localização.

De modo a possibilitar que o framework possa ser usado com vários tipos de tecnologia de localização, foi criado um ponto adaptável através da interface *PositioningAdapterIF*, com o uso do padrão de projeto *Adapter*. As classes adaptadoras, que implementam essa interface, servem de fachada entre a tecnologia de localização e o framework. Na implementação do método *registerUser,* deve-se registrar um cliente móvel na tecnologia de localização. Esse método possui como argumento, além do identificador do usuário, um objeto de configuração do tipo *Object*, que encapsula qualquer informação que a tecnologia de localização necessita para identificar um dispositivo móvel. Por exemplo, o serviço *LIS*, descrito mais à frente, necessita apenas do *MAC address*[5] do dispositivo.

A obtenção da posição de um dispositivo pode ser feita de dois modos: *pull* ou *push*. No primeiro, a tecnologia de localização é explicitamente inquirida, através do método *getPosition* da classe adaptadora, sobre a posição de um dispositivo previamente registrado. Já no modo *pull,* é a classe adaptadora que notifica a classe *Positioning,* chamando o método *setUserPosition* desta, quando for detectado que um dispositivo mudou sua posição. Essa notificação é feita automaticamente em um tempo definido pela própria tecnologia de localização. Se a tecnologia de localização, entretanto, já informar a região lógica e não apenas uma informação de localização "bruta" (coordenadas geográficas, intensidade de sinal, etc.), o método a ser chamado é *setUserLocation.* A diferença entre os dois é que em *setUserPosition, Positioning* primeiro consulta o gerenciador de regiões simbólicas para que este traduza a informação de posição para uma região lógica. Em *setUserLocation* isso não é necessário.

[5] *Medium Access Control address* é o endereço da interface de rede do dispositivo móvel.

Há que se ressaltar que essa interface serve mais para as chamadas tecnologias de localização centradas na rede *(net-centric)*. Isso porque as tecnologias centradas no aparelho *(device-centric)* informam a posição do usuário ao próprio programa cliente, que envia a informação para o programa servidor. Neste caso, a classe que implementa a interface *RouterIF* deveria encaminhar esse dado diretamente para *Positioning*, chamando seu método *setUserPosition.* Já as centradas na rede entram em contato com o programa servidor através das classes adaptadoras.

O framework oferece como implementação padrão de *PostioningAdapterIF* a classe *LisAdapter,* que serve de fachada para o *Location Inference Service (LIS)* (Rubinsztejn et al., 2004) da arquitetura *MoCA. LIS* é uma tecnologia de localização que infere a posição de um dispositivo móvel através da análise da intensidade do sinal de redes sem-fio 802.11. No LIS também é possível definir regiões retangulares e de tamanho arbitrário, atribuir um nome para estas e construir uma topologia hierárquica a partir destas regiões simbólicas.

As tecnologias de localização geralmente informam a posição de uma forma "bruta", que não são úteis para muitas das aplicações baseadas em localização. Assim, faz-se necessário que haja uma tradução dessa posição para nomes das localidades correspondentes (nomes simbólicos) O framework possibilita tal conversão através de um ponto adaptável representado pela interface *SymbolicRegionManagerIF.* Seu método *translate* deve ser implementado de forma a receber como parâmetro a informação de posição do usuário e retornar uma string que indica o nome simbólico correspondente. Além disso, devem informar, através dos seus métodos *getSymbolicRegions,* o nome de todas as regiões simbólicas mapeadas. *Positioning* também funciona como fachada para acessar tais métodos por outros objetos do sistema. Dessa forma, eles não precisam conhecer objetos *SymbolicRegionManagerIF*.

Quando o método *setUserPosition* de *Positioning* é invocado, este chama *translate* em *SymbolicRegionManagerIF,* que recebe a posição como parâmetro e retorna a região simbólica correspondente. Então, *Positioning* guarda essa informação numa tabela *hash* indexada pelo identificador do usuário e notifica os objetos interessados na mudança de localização de tal usuário. Essa tabela é consultada quando outros objetos precisarem saber a posição de um indivíduo.

Na atual versão, o framework lida somente com áreas (localidades) atômicas ao invés de áreas hierárquicas. Assim, uma possível extensão para *SymbolicRegionManagerIF* seria a criação de uma classe que oferecesse métodos para o acesso a uma hierarquia de áreas, isto é, que seria composta de áreas aninhadas como, por exemplo, um prédio composto de vários pisos, cada qual composto de um corredor e salas.

Qualquer objeto pode registrar-se através do método *addPositionListener* para ser notificado da mudança de localização de um usuário móvel. Basta que implemente a interface *PositionListenerIF*, que tem um único método *locationChanged,* o qual é chamado por *Positioning* quando um usuário muda a sua localização. O método recebe como parâmetro o identificador do usuário e uma string informando o nome da sua nova localidade. Essa notificação foi implementada seguindo o padrão de projeto *Observer*, provendo, dessa forma, um desacoplamento entre as classes notificadas e a notificadora.

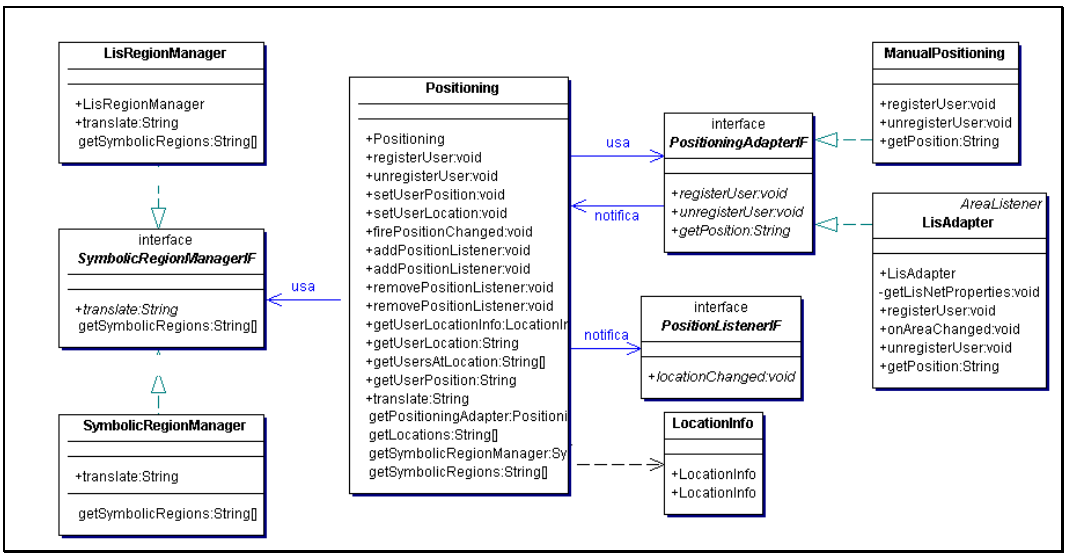

Figura 3 – Diagrama de classes do pacote *flocs.positioning.*

Outros objetos do sistema também podem consultar a localização de um usuário sem ter que esperar pela notificação, através do método *getUserLocation* de *Positioning*. Se desejarem, podem também chamar o método *getUserLocationInfo,* que retorna um objeto encapsulando, além da localização, o momento em que esta foi atualizada pela última vez. Essa informação temporal é muito útil quando a posição não é atualizada constantemente. Por exemplo, em um sistema que utiliza posicionamento manual, isto é, quando a localização é

informada pelo próprio usuário, como é o caso do programa *BuddySpace* (ver Cap. 2), um usuário pode decidir se a localização dos seus amigos mostrada no mapa é confiável ou não com base no tempo que a informação foi atualizada.

A Figura 3 ilustra o diagrama de classes deste pacote.

3.3.3. Localidades

O pacote *flocs.location* contém as classes relacionadas com a representação computacional da região geográfica onde se encontra o usuário do dispositivo móvel. Cada região geográfica é mapeada para uma região simbólica, representada por objetos da classe *Location,* os quais guardam informação sobre quais usuários encontram-se nelas e as mensagens postadas ali.

As localidades podem ser configuradas para que verifiquem o tempo de vida de suas mensagens. Cada localidade tem um objeto *LocationConf*, que guarda tais configurações. Seu atributo booleano *checkExpiration* permite definir se a localidade vai verificar a data de expiração das mensagens. Se for configurado para falso, todas as mensagens postadas na localidade ficarão indefinidamente nela. Caso contrário, a tarefa de monitoramento da expiração é delegada a um agente de software, da classe *ExpirationAgent,* que consulta os outros atributos de *LocationConf* para executá-la.

Para restringir o tempo de vida máximo que uma mensagem pode permanecer na localidade, configura-se o atributo *expiration,* que recebe valores inteiros correspondendo a minutos. Por exemplo, se a validade deve ser de um dia, o atributo deverá receber o valor de *24*60 minutos.* Por questão de desempenho, pode-se configurar o atributo *intervalExpirationCheck* para definir de quantos em quantos minutos o agente deve verificar a lista de mensagens da localidade, para excluir aquelas que estiverem expiradas. Se um desenvolvedor achar necessário, pode estender a classe *LocationConf* e criar subclasses com configurações extras.

Localidades podem receber objetos derivados da classe abstrata *Message,* que entre outros, possui os seguintes atributos: autor, assunto, um identificador da mensagem, o modo de expiração, etc. Como os objetos postados em uma localidade precisam ser persistidos, *Message* implementa *java.io.Serializable.*

Quanto à expiração, uma mensagem pode ser de três modos: permanente, expirável e instantânea, que são definidas pelas constantes *PERMANENT, EXPIRABLE* e *INSTANTANEOUS* da classe *Message*. Caso seja permanente, a mensagem terá tempo de vida igual ao tempo máximo permitido pela localidade em que foi postada. Se for expirável, o *ExpirationAgent* verificará seu tempo de vida, e será apagada da localidade assim que este se esgotar. Por fim, se for instantânea, a mensagem não é guardada na localidade. Esta apenas notifica os objetos registrados de que uma mensagem foi postada na região, passando-a para eles. Portanto, nos dois primeiros casos, a mensagem é usada para comunicação assíncrona entre os clientes, e no terceiro caso, para comunicação síncrona.

O único método abstrato dessa interface é *getMessageType,* que as subclasses devem implementar para retornar o tipo do objeto. Outros métodos além daqueles oferecidos por *Message* também podem ser implementados nas suas subclasses para proporcionar funcionalidades extras.

Quando uma mensagem é postada em uma localidade, o objeto *Location* correspondente verifica se o tempo de vida dela possui um valor acima do permitido na localidade. Se for este o caso, muda para o valor limite. Além disso, sua data e hora (*timestamp*) de chegada é configurada através do seu método *setDate.* Isso porque não seria exato usar a hora em que a mensagem foi criada no cliente, pois seu relógio pode não estar sincronizado com o do servidor, o que causaria inconsistência com a tarefa de verificação de expiração. Com esse atributo configurado, o agente pode obter, a partir do método *getExpirationDate* da própria mensagem, a data/hora exata em que a mensagem expira. Esse método funciona somando ao *timestamp* da mensagem os minutos definidos no seu atributo *ttl (time to live*, tempo de vida em inglês), retornando um objeto do tipo *java.util.Calendar.*

Já que as mensagens postadas em uma localidade podem ser de vários tipos, elas são guardadas em forma de listas em uma tabela *hash* indexadas por seus tipos. Assim, no momento em que são postadas, a localidade verifica o seu tipo e coloca na lista correspondente. Essa separação permite que se obtenha apenas uma única classe de mensagens. Por exemplo, quando um usuário portando um dispositivo com pouca memória entrar em uma região, ao invés de receber todos os tipos de mensagens, este pode escolher receber apenas textos, que ocupam menos largura de banda e memória do que mensagens que são imagens.

Outros objetos podem ser notificados quando mensagens ou usuários são adicionados/removidos de uma localidade. Para isso, basta que implementem as interfaces *MessageInLocationListenerIF* e *UserInLocationListenerIF,* respectivamente, e registrem-se no objeto *Positioning.* Assim, receberão uma referência para os usuários/mensagens que foram adicionados/removidos e a localidade onde o evento ocorreu.

A classe *LocationManager (LM)* controla o ciclo de vida dos objetos *Location*, sendo responsável pela instanciação destes objetos, bem como pela recuperação e salvamento das configurações e objetos postados em cada localidade. Geralmente, os outros objetos do sistema não têm acesso direto a objetos *Location,* mas sim ao nome que identifica a localidade. Assim, consultam *LM* passando o nome da localidade através de seu método *getLocation* para obter uma referência ao objeto correspondente. Além disso, *LM* funciona também como fachada para os objetos *Location*, possuindo métodos para postar mensagens e adicionar usuários a uma localidade, sendo necessário apenas informar seu nome.

A Figura 4 ilustra o diagrama de classes deste pacote.

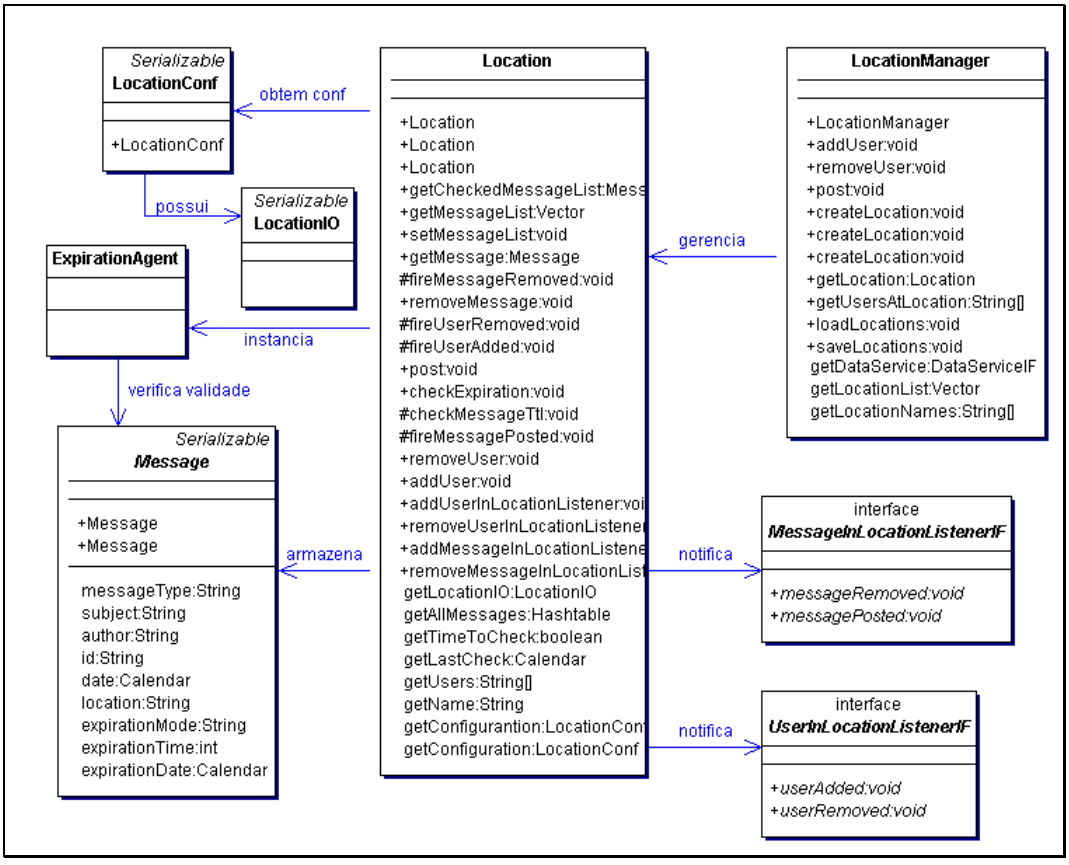

Figura 4 – Diagrama de classes do pacote *flocs.location.*

3.3.4. Comandos

O pacote *flocs.command* contém vários comandos que podem ser usados para trocar informações entre a aplicação cliente e a servidora. Um comando é um objeto que carrega atributos necessários para executar determinada ação no sistema. Por exemplo, para fazer um login existe o comando *Login* que possui como atributos o login do usuário, sua senha e suas configurações de rede (IP, porta, placa de rede, etc), estas últimas necessárias para que o servidor entre em contato com o cliente.

Todos os comandos desse pacote têm como superclasse *Command*, que implementa *java.io.Serializable*. Dessa forma, suas subclasses já herdam essa interface. Entretanto, não é necessário que novos comandos sejam obrigatoriamente subclasses de *Command.* A parte de comunicação do framework é flexível, pois envia e recebe objetos do tipo *Object,* superclasse de todos os objetos JAVA. Portanto, os comandos criados pelo desenvolvedor da aplicação podem ser de qualquer tipo. Vale ressaltar também que esses comandos podem ser estendidos e dessa forma terem novos atributos.

Como ilustração, a Tabela 2 resume alguns comandos implementados no framework.

Comandos	Atributos	Origem	Uso
Command	-	-	Superclasse de todos os comandos
CreateUser	nome, login, senha	Cliente	Cria usuário no sistema
ErrorMessage	código, mensagem, objeto, destino	Servidor	Informa a ocorrência de uma exceção
LocationChange	login, nova localidade, mensagens postadas	Servidor	Informa mudança na localização do cliente
LocationCommand	login, localidade	Cliente	Cliente informa mudança de sua localização (posicionamento manual)
Login	login, senha, endereço de rede (ip, porta)	Cliente	Login no sistema
LoginResponse	localização corrente, lista de localidades, conectado?, mensagem	Servidor	Resposta ao login
Logout	login, senha	Cliente	Logout do sistema
MessageCommand	mensagem, localidade-destino	Cliente	Envio de mensagem a localidades
SimpleMessage	mensagem	Servidor	Mensagens informativas

Tabela 2 – Comandos oferecidos pelo framework.

3.3.5. Dados

O pacote flocs.data contém as classes responsáveis pela persistência de dados. A interface *DataServiceIF (DS)* foi criada como um ponto de adaptação que possibilita ao framework utilizar várias formas de persistência. Essa interface tem apenas os métodos que são utilizados por classes do próprio framework para salvar dados, tais como as configurações das localidades e os perfis dos usuários. Entretanto, métodos adicionais podem ser criados pelo desenvolvedor de aplicações ao implementar essa interface.

Geralmente, não há necessidade de salvar objetos inteiros, mas apenas as informações necessárias para a reconstrução destes quando forem carregados da base de dados. Assim, as classes *Location* e *User* possuem métodos que retornam, respectivamente, objetos *LocationIO* e *UserIO*, cujos atributos têm somente essas informações essenciais. No sentido inverso, cada uma dessas classes possui um construtor que recebe objetos desses tipos como parâmetro e conseguem reconstruir o objeto original. Sendo assim, *DS* tem apenas métodos genéricos que salvam e carregam objetos *LocationIO* e *UserIO*. Cabe às classes que a implementam a tarefa de persistir tais objetos.

A implementação oferecida pelo framework persiste dados através de arquivos em disco por meio da classe *FileService*. Esta utiliza a classe auxiliar *FileIO*, que possui métodos genéricos que salvam e carregam objetos em arquivos. Desenvolvedores de instâncias que necessitem gravar outros tipos de objetos podem utilizar essa classe, desde que obedeçam à restrição da linguagem JAVA, que obriga que os objetos a serem gravados em disco sejam serializáveis. Por isso, *LocationIO* e *UserIO* implementam *java.io.Serializable.*

3.3.6. Classes auxiliares

No pacote *flocs.auxiliar* encontram-se as classes auxiliares do framework.

- **Usuário**

A classe *User* é utilizada para guardar informações sobre os usuários do sistema, como nome, login, senha, informações de rede, localidade corrente e etc.

As classes do framework não a utilizam diretamente, pois todas as referências a um usuário são feitas através de uma string identificadora deste, como, por exemplo, seu login. Entretanto, a idéia é que as instâncias do framework estendam essa classe e a utilizem como um proxy para o cliente, executando tarefas de interesse deste. Por exemplo, no programa *BuddySpace* (ver Cap. 4), um objeto *User* registra-se em *Positioning* para ser notificado da mudança de localização dos usuários que se encontram na lista de amigos do cliente que representa. Além disso, *User* pode ser usado para filtrar, adaptar ou guardar mensagens quando seu cliente encontrar-se desligado ou fora da área de cobertura da rede sem fio.

- **Perfis**

Cada usuário tem um perfil associado a ele, implementado no sistema pela classe *Profile*. Ela guarda a lista de amigos e uma lista com as mensagens que já foram lidas pelo usuário. Esta última é útil para evitar que este receba a mesma mensagem repetidas vezes. Assim, sempre que um usuário tiver lido uma mensagem, o método *setRead* deve ser chamado, passando como parâmetro a mensagem ou conjunto de mensagens lidas. No caminho inverso, para saber se uma mensagem foi lida, basta chamar o método booleano *hasRead* juntamente com o identificador da mensagem.

O problema com essa estratégia é que a lista vai acumulando todas as mensagens recebidas, tornando essa consulta cada vez mais lenta. Para contornar esse problema, à medida que a lista de mensagens é varrida, a data de expiração das mensagens é checada. As que tiverem expirado são apagadas. Tais mensagens somente estavam ocupando espaço, pois com certeza já foram deletadas da localidade pelo agente que verifica o tempo de expiração e nunca entrarão como parâmetro na consulta para verificar se já foram lidas, quando um usuário entrar numa localidade.

Criando-se uma classe que estenda *Profile* esse comportamento pode ser modificado e novas funcionalidades podem ser acrescentadas. Depois, basta passá-la como parâmetro para o método *setProfile* de *User*. Dessa forma, o perfil também se torna um ponto adaptável do framework. Vale lembrar que, como os perfis são persistidos, a subclasse também deve ser serializável.

- **Registro de componentes**

Algumas classes do framework, como *LocationManager*, *Positioning* e *CommunicationService*, têm apenas uma única instância no sistema que deve ser acessada por todas as outras classes. Uma boa solução para evitar que essa instância única tenha que ser passada no construtor das classes que as utilizam é implementá-la com o padrão de projeto *Singleton*. Entretanto, o uso desse padrão não permite que sejam criadas subclasses das mesmas, pois seu construtor não pode ser instanciado, já que é privado. Assim, desenvolvedores que quisessem, por exemplo, acrescentar um novo comportamento ao gerenciador de localidades, não poderiam estender a classe *LocationManager*, já que deve existir apenas uma instância dela no sistema e essa seria implementada como um *Singleton*.

Para contornar esse problema, foi criada a classe *ComponentRegistry(CR),* cuja função é centralizar o acesso às classes de instâncias únicas. *CR* cria uma única instância dessas classes e tem métodos que possibilitam às outras classes acessarem essa instância. Tomando o exemplo de *LocationManager(LM)*, se um desenvolvedor quiser outro gerenciador de localidades, basta instanciar uma subclasse de *LM* e passá-la ao método *setLocationManager* de *CR*.

Um ponto a ressaltar é que apesar das classes *LocationManager, Positioning* e *CommunicationService* serem pontos fixos, *CR* cria uma forma de transformá-las também pontos adaptáveis, tornando o framework ainda mais flexível. Além disso, no projeto dessas classes, tomou-se o cuidado de criar seus métodos e atributos como *protected* e não *private,* de modo a serem visíveis pelas suas subclasses.

CR é também o lugar onde o desenvolvedor deve registrar os pontos adaptáveis do framework, através dos diversos métodos *set<Nome>* de *CR*. Todos os *hot spots* do framework tem classes padrões implementadas. *CR,* já no seu construtor, instancia essas classes e as liga aos *frozen spots* que as utilizam. Dessa forma, o desenvolvedor tem somente o trabalho de registrar os *hot spots* que ele deseja alterar, pois os outros já foram previamente registrados por *CR*.

Mais uma vez, vale ressaltar que todo acesso às classes de instância única do sistema deve ser feito somente a partir de *CR* sob pena de existirem várias instâncias criadas, o que tornaria o programa inconsistente.

3.3.7. Dinâmica do Sistema

Depois da descrição detalhada de cada componente do framework, nesta seção descreveremos três freqüentes casos de uso do framework, comuns também a várias aplicações para comunicação baseada em localização encontradas na literatura, e que ilustram como as diferentes partes do framework interagem.

São eles:

i) Postagem de uma mensagem em uma localidade;

ii) Recebimento de uma mensagem postada; e

iii) Mudança de localização de um usuário.

Esses casos de uso são ilustrados nas Figuras 5, 6 e 7, respectivamente.

No primeiro, o usuário escreve uma mensagem na aplicação cliente que executa no dispositivo móvel, ajusta seu tempo de expiração e executa o comando para postá-la. A aplicação cria um objeto *Message,* colocando o texto, a localidade e o autor da mensagem, e o coloca em um comando do tipo *MessageCommand*. Passa-o, então, ao método *send* de *CommunicationService.* Este, solicita a *CommunicationFactoryIF* o objeto *CommunicationIF* padrão. Como o endereço do *host* destino supostamente já havia sido anteriormente registrado através do seu método *registerDestination*, *CommunicationIF* abre uma conexão com o dispositivo e envia o comando pela rede sem fio.

No lado servidor, *CommunicationIF* obtém o comando da rede fixa e notifica *RouterIF*, chamando seu método *onCommand. RouterIF* analisa o tipo de comando, verifica que é do tipo *MessageCommand* e o passa para o método *post* de *LocationManager*. Este obtém o objeto *Location* que representa a localidade informada pelo comando e posta o objeto *Message* chamando o método *post.* Em *Location*, um *timestamp* é colocado na mensagem. Verifica-se se a mensagem é síncrona. Se não for, checa-se se o tempo de vida é maior do que o estabelecido em *LocationConf*. Se for, o *timestamp* é modificado. *Location* verifica, então, que o tipo da mensagem é *TEXT* e a coloca na lista de mensagens de texto. Essa e as demais listas são constantemente verificadas por *ExpirationAgent,* que exclui as mensagens cujo tempo de vida expirou. Por fim, *Location* varre sua lista de *MessageInLocationListenerIFs,* notificando esses objetos de que uma mensagem foi postada na localidade. Se a mensagem fosse síncrona, ela não seria guardada

na localidade. *Location* apenas notificaria sua lista de *listeners* de que uma nova mensagem foi postada na região.

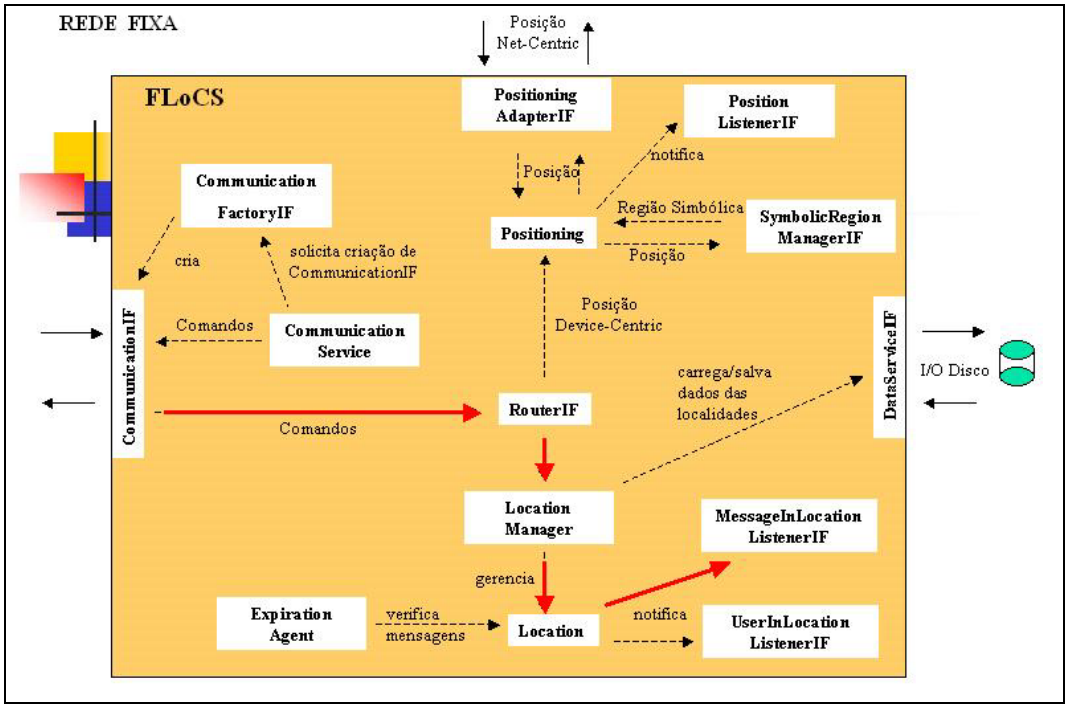

Figura 5 – Postagem de uma mensagem em uma localidade.

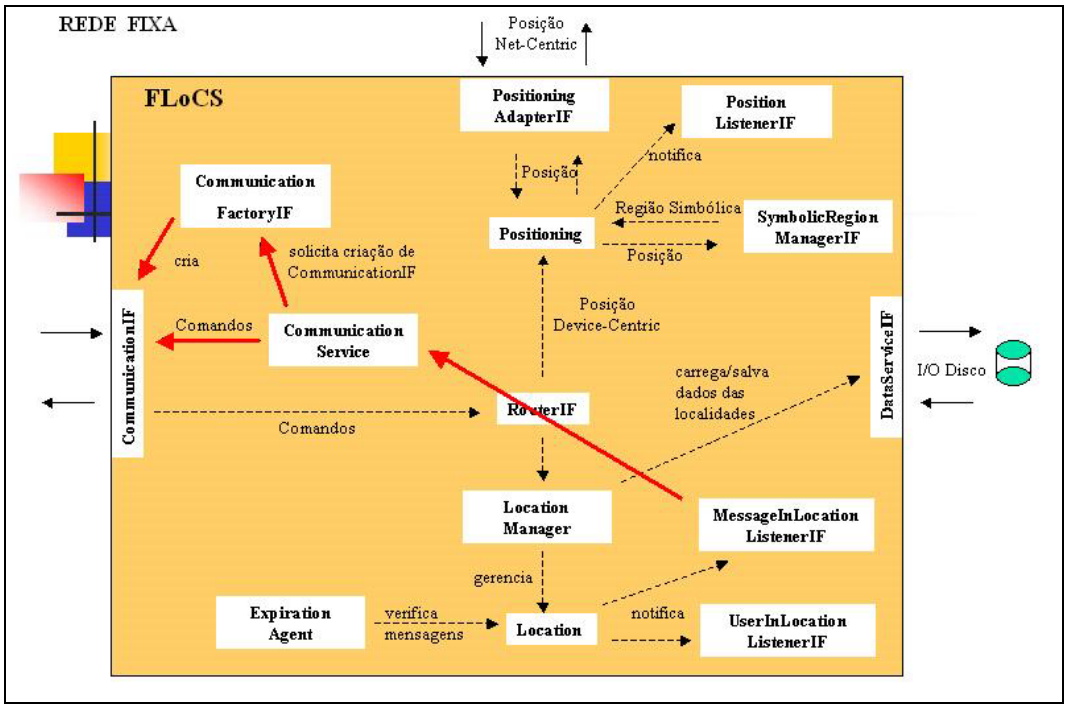

Figura 6 – Recebimento de uma mensagem postada.

O segundo caso de uso – o recebimento de uma mensagem postada (Fig. 6) - começa onde termina o anterior. Digamos que um objeto *User* tenha implementado a interface *MessageInLocationListenerIF*. Ele é notificado por *Location* que lhe passa o objeto *Message* postado nela. Como é uma mensagem postada recentemente, *User* não necessita consultar seu objeto *Profile* para verificar se a mensagem já foi lida pelo usuário que ele representa. Cria, então, um comando *MessageCommand,* onde coloca a mensagem, e chama o método *send* em *CommunicationService* passando o identificador do usuário como destino e o comando como dado. *CommunicationService* solicita a *CommunicationFactoryIF* o objeto *CommunicationIF* padrão. Como supostamente o destino já tinha sido anteriormente registrado, *CommunicationIF,* abre uma conexão com ele e lhe envia o comando pela rede.

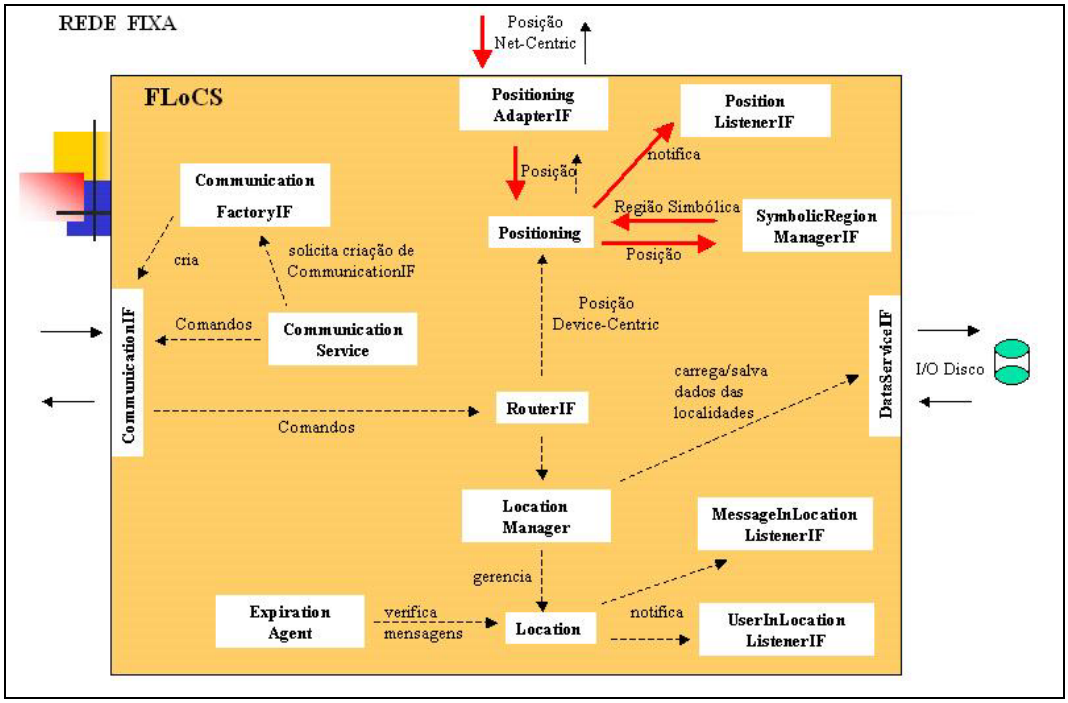

Figura 7 – Mudança de localização de um usuário.

O caso de uso - mudança de localidade (Fig. 7) - inicia com a tecnologia de localização (neste exemplo, ela é do tipo *net-centric*) notificando *LocationAdapterIF* de que um usuário mudou sua posição. Este chama o método *setUserPosition* em *Positioning,* passando o identificador do usuário e a sua nova posição, que por sua vez chama *translate* em *SymbolicManagerIF* para que este informe a localidade que a posição representa. Depois, *Positioning* consulta sua

lista de usuários registrados e obtém a localidade corrente do usuário em questão. Se ela for igual à nova, o usuário não mudou de localidade e, então, *Positioning* não faz mais nada. Do contrário, atualiza a nova localização do usuário e varre sua lista de objetos *PositionListenerIF*, notificando-os de que aquele usuário mudou de localidade.

O próprio usuário que mudou de localização só vai saber desse fato se o objeto *User* que o representa no sistema tiver se registrado em *Positioning* para receber notificações sobre esse evento. Quando for notificado, *User* cria um comando *LocationChanged* e o envia para o seu cliente através de *CommunicationService.*

Digamos que um objeto *User* represente um outro usuário em cuja lista de amigos figura o usuário que acaba de mudar de localidade. *User* já tinha se registrado anteriormente em *Postioning* para receber notificações sobre mudanças de localização desse amigo. Quando esse evento ocorre, *Postioning* notifica *User* que cria um comando com essa informação e envia à aplicação cliente do usuário que representa, como explicado no caso de uso anterior.

3.4. Instanciação do framework

O framework é voltado primariamente para a instanciação do servidor da aplicação, não impondo nenhuma arquitetura que deva ser seguida no desenvolvimento do cliente. Entretanto, alguns serviços oferecidos, tais como comunicação, persistência de dados e comandos, podem ser reutilizados na programação do cliente da aplicação. Cabe então ao desenvolvedor criar a interface gráfica e a lógica para acesso a tais serviços.

Para instanciar uma aplicação a partir do framework, o primeiro passo é verificar quais são seus requisitos em termos de:

- *Tipo de Comunicação:* protocolo a ser utilizado, orientado a conexão ou datagrama, etc;
- *Informação de localização:* forma de obtenção da localização de um usuário, tecnologia de localização a ser utilizada, forma de mapeamento de regiões físicas em lógicas;

- *Localidades:* tipos dos objetos postados, características das localidades, como, por exemplo, se deve haver controle de expiração de mensagens, e etc;
- *Persistência de dados:* utilização de um banco de dados ou sistema de arquivos;
- *Comandos específicos da aplicação:* tipos de mensagens trocadas entre os clientes e o servidor;
- *Usuário:* representação do usuário no sistema, perfil, lista de amigos e etc.

De posse dessas informações, deve-se verificar se as implementações dos pontos adaptáveis oferecidas pelo framework satisfazem os requisitos da aplicação. Se alguma não satisfizer, a interface referente a esse ponto adaptável deve ser implementada. Essas novas implementações devem obrigatoriamente ser registradas em *ComponentRegistry,* dentro da classe principal da aplicação.

A Tabela 3 apresenta os pontos adaptáveis, separados por pacote, resumindo as implementações oferecidas pelo framework.

Pacote	Ponto Adaptável	Implementação	Descrição
communication	CommunicationFactoryIF	CommunicationFactory	Cria objetos dos tipos *TcpSocket* e *MocaPubSub*
	CommunicationIF	TcpSocket	Comunicação por sockets TCP
		MocaPubSub	Interface para serviço de eventos *ECI/MoCA*
	RouterIF	UselessRouter	Apenas para fins ilustrativos/testes
positioning	PositioningAdapterIF	LisAdapter	Interface para tecnologia de localização *LIS/MoCA*
		ManualPositioning	Interface para entrada manual da localização
	SymbolicRegionManagerIF	SymbolicRegionManager	Apenas para fins ilustrativos/testes
		LisRegionManager	Interface para tecnologia de localização *LIS/MoCA*
location	Message	-	Classe abstrata
	LocationConf	-	Classe concreta funcional
data	DataServiceIF	FileService	Persistência de dados em arquivos
command	Command	(Ver tabela da Seção 3.3.4)	
auxiliar	Profile	-	Classe concreta funcional
	User	-	Classe concreta funcional

Tabela 3 – Implementações de pontos adaptáveis providas pelo framework.

O passo seguinte é analisar quais os eventos gerados pelo framework são de interesse da aplicação. Então, deve-se registrar os objetos que vão tratá-los nas classes geradoras de eventos, lembrando de implementar as interfaces requeridas.

Por fim, deve-se implementar os outros aspectos específicos a cada aplicação, tais como o fluxo dos comandos e as classes que vão tratá-las, a necessidade de criar novas classes com funcionalidades extras não contempladas no framework, e etc. Uma boa prática para a definição do fluxo de comandos e mesmo a criação destes é analisar os casos de uso da aplicação, bem como os diagramas de seqüência que os descrevem.

No próximo capítulo, descreveremos duas aplicações instanciadas seguindo os passos definidos nesta seção.